

Evaluating Hash Tables for Complex Graph Algorithms: A Parallel Dynamic Community Detection Case Study

Gabriel G. dos Santos¹, César A. F. De Rose¹, Kaio M. Vieira², Vinicius dos Santos²

¹Escola Politécnica – PUCRS
Porto Alegre – RS – Brazil

²Departamento de Ciência da Computação – UFMG
Belo Horizonte – MG – Brazil

`gabriel.giordani@edu.pucrs.br, cesar.derose@pucrs.br,`
`{kaiovieira,viniciussantos}@dcc.ufmg.br`

Abstract. *Community Detection is a widely studied problem in the context of static and dynamic graphs due to its wide range of applications. Although parallel algorithms have been investigated for such problems, the choice of the underlying structures used by implementations receives little spotlight. Dynamic community detection requires repeated insertion, deletion, and access of multiple structures as the graph evolves, making the containers a first-order performance concern. This paper evaluates two hash table implementations as containers for vertices, communities, and edges inside DynaMap, a parallel dynamic community detection algorithm. Through experiments on four real-world graphs, we demonstrate that no single hash table configuration is best for every structure.*

1. Introduction

Graphs are a widely used data structure that can easily represent the relationship between data points. The structure is comprised of a set of vertices and a set of edges, where each edge connects two vertices. Static graphs assume an unchanging structure, i.e., updates to the graph are not considered and, therefore, are not provided. However, many graphs of practical interest – social networks, communication traces, transportation systems, biological interaction networks – are not static objects but evolve continuously, with vertices and edges being inserted and removed as time passes. Algorithms that operate on such graphs must therefore be designed not only to answer queries efficiently, but to *maintain* their internal state under a stream of updates. This requirement has motivated a growing body of systems work on dynamic graph data structures, ranging from block-based streaming representations [Ediger et al. 2012] to hierarchical containers that adapt their layout to the degree distribution of the graph [Pandey et al. 2021], to hash-indexed adjacency structures designed explicitly around fast point updates [van der Grinten et al. 2022, Fan et al. 2025]. Dynamic community detection is a representative case: each edge insertion or deletion can trigger cascading updates to community assignments, so the algorithm’s inner loop is dominated not by traversal of a fixed structure, but by repeated membership tests, lookups, and in-place modifications of several auxiliary associative structures.

Hash tables are a natural fit for this access pattern. Sorted arrays or compressed adjacency representations offer excellent query performance but amortize poorly

over frequent single-element updates. Balanced-tree or packed-array structures guarantee good asymptotic update bounds but carry higher constant factors on the point queries and point updates that dominate this workload. On the other hand, a well-tuned hash table offers expected $O(1)$ insertion, deletion, and lookup with comparatively little per-operation overhead. This is precisely why hashing, or hashing-adjacent designs, recur throughout the dynamic-graph systems literature as a building block for representing adjacency, membership, and auxiliary bookkeeping information under mutation [Ediger et al. 2012, van der Grinten et al. 2022, Fan et al. 2025]. At the same time, “a hash table” is not a single design point: the choice of collision-resolution strategy, memory layout, and hashing policy materially affects performance, and a single configuration is rarely optimal across every associative structure a graph algorithm needs to maintain.

In this paper, rather than proposing a new dynamic graph container, we take a systematic, empirical look at how far this choice matters when the container implementation is otherwise fixed. We evaluate two hash table designs from the GNU libstdc++ Policy-Based Data Structures (pb_ds) library – `cc_hash_table`, which resolves collisions by chaining, and `gp_hash_table`, which resolves them via open addressing, evaluated with both linear and quadratic probing – as drop-in components of a dynamic community detection algorithm. We use these containers to store vertices, edges, and communities. Unlike common workloads, our algorithms require the construction of super-graphs, which makes graph construction a process as relevant as graph data access. Therefore, deciding the best manner to represent each component is not necessarily straightforward. Our results show that no single configuration dominates across all three roles inside the same dynamic community detection algorithm: `cc_hash_table` is the better choice for the vertex and community structures, while `gp_hash_table` is the better choice for the edge structure. Furthermore, our experiments show the bottlenecks that different hash table designs incur, providing a useful guide to selecting such designs in more complex scenarios.

This paper is organized in the following manner: Section 2 presents the necessary background on data structures for dynamic graphs, the hash table schemes utilized in this paper, and the overview of the algorithms used; Section 3 presents the experiments with the hash tables from the GNU pb_ds library; Section 4 discusses the experiment results and the main findings; Section 5 presents related work; and Section 6 concludes this paper.

2. Background

Due to the increased rate of data generation today, many graph applications require a transition to a dynamic graph model. Static graph processing becomes sub-optimal once updates arrive faster than the algorithm can process the graph, making results invalid for the new graph state. An essential part of dynamic graph processing is the storage and access of graph data. The most common structure for static graph processing engines, the CSR, was not designed to receive updates. Therefore, multiple new data structures tailored for dynamic graphs have been proposed over the years. Such structures try to balance efficient mutations with fast element access.

In the remainder of this Section we explore some of the data structures proposed

for dynamic graphs, the hash table schemes and policies we experiment with in this paper, and the community detection algorithms we used as a use case.

2.1. Dynamic Graph Representations

A common approach is to separate the graph elements into independent blocks [Ediger et al. 2012, Feng et al. 2015, Green and Bader 2016, Dhulipala et al. 2019, Firmli and Chiadmi 2023]. Whenever changes need to be made to any element of a block, only that block needs to be changed, leaving the rest of the data structure unchanged. STINGER [Ediger et al. 2012] and its distributed version [Feng et al. 2015] use linked lists of blocks. Aspen [Dhulipala et al. 2019] uses a C-tree, which is a compressed purely-functional tree that stores blocks as tree nodes. CSR++ [Firmli and Chiadmi 2023] uses an array of blocks to store vertices, and each vertex has an array that contains the block and ID of each of its neighbors.

Some works choose to use hash tables to store the graph [Iwabuchi et al. 2016, Jaiyeoba and Skadron 2019, Zhang et al. 2020]. DegAwareRHH [Iwabuchi et al. 2016] separates the vertices into two hash tables depending on their degree. Low-degree vertices are stored with their properties and edges, while middle- and high-degree vertices store their edges on another hash table. GraphTinker [Jaiyeoba and Skadron 2019] stores the edges in a balanced hash table, while the vertices are stored in a compressed list.

2.2. `cc_hash_table` and `gp_hash_table`

The C++ standard requires that references and pointers to elements remain valid across insertions, which in practice forces conforming `unordered_map` implementations to use chaining with individually heap-allocated nodes [GNU Project 2024]. However, if the algorithm using such hash table does not need to hold long-lived references across mutation, this guarantee is unnecessary overhead. On the other hand, the GNU libstdc++ Policy-Based Data Structures (`pb_ds`) library provides hash table implementations that are more configurable and typically faster, making them an attractive substitute.

`cc_hash_table` is a *collision-chaining* hash table: each bucket holds a linked list of colliding elements, closely mirroring the architecture of `std::unordered_map`, but exposing the hash-combining strategy, resize policy, and hash-caching behavior as template policies rather than hard-coding them [GNU Project 2024]. `gp_hash_table` is a *general-probing* hash table: it uses open addressing, storing elements directly, inline, in a single flat array, and resolving collisions by probing for the next available slot according to a configurable probe sequence [GNU Project 2024]. This difference in storage layout has direct performance consequences. Chaining incurs a pointer indirection and a separate heap allocation per element, but degrades gracefully as the load factor grows, since an arbitrarily long chain can always accommodate more collisions. Open addressing avoids per-element allocation entirely and packs elements contiguously, which is considerably friendlier to the cache, but requires maintaining a lower load factor and a bounded probe sequence to keep lookups fast, since every occupied slot competes for space in the same underlying array.

Linear probing searches the sequence of slots $h(k), h(k)+1, h(k)+2, \dots$ (modulo the table size). It is the most cache-friendly option since successive probes touch adjacent memory locations. However, it is also the most susceptible to *primary clustering*: once

a run of occupied slots forms, any key hashing into that run extends it further, so probe lengths grow disproportionately as the load factor increases or as insertions arrive in a skewed order.

Quadratic probing searches a sequence of the form $h(k), h(k) + 1^2, h(k) + 2^2, \dots$, which spreads colliding keys across positions that are increasingly far from the original slot. This breaks up primary clustering at the cost of touching non-contiguous memory more often, trading sequential locality for a probe sequence that stays short more consistently as the table fills or as the input becomes skewed.

2.3. Dynamic Community Detection with DynaMap

We choose to use DynaMap [Santos et al. 2025] as the dynamic community detection algorithm to be evaluated with different hash tables. DynaMap is a dynamic variant of the Infomap algorithm [Rosvall et al. 2009], which is widely used for static community detection. The core of DynaMap consists of creating intermediary structures for communities affected by updates and then running Infomap on such structures, refining the community assignments. This reduces redundant computation that would be created if Infomap were executed from scratch.

Infomap itself is an agglomerative community detection algorithm that uses the Map Equation [Rosvall et al. 2009]. This means that an initial clustering is done by moving vertices to neighboring communities until it is not possible to further optimize the Map Equation. Afterwards, Infomap creates a super-graph by transforming each community into a super-vertex and runs its clustering again on this new graph. This process continues until there is a super-graph that is unable to improve the Map Equation. Once the clustering process terminates, a refinement step is executed to correct community assignments that might have become suboptimal.

2.3.1. Implementation and Parallelization Details

We utilize DynaMap’s codebase¹ for all experiments. There is one hash table for storing vertices and one for storing communities. Each vertex has a hash table for storing its edges.

To effectively parallelize the movement phase of Infomap, a community must be updated by at most one thread at a time. Therefore, we employ a community-level locking mechanism to make vertex movements. Whenever a vertex needs to be moved between communities, the thread responsible for the movement must first acquire a lock for both communities. Accessing information on vertices and edges can be done concurrently, and accessing information about communities requires atomic clauses.

The process of creating super-graphs has an inherently sequential step of creating the initial entries in the hash table. After each super-vertex entry is created, we can populate the information (edges, weights, children) in parallel with other super-vertices.

¹<https://github.com/GGRV-PUCRS/DynaMap>

3. Experiments

To evaluate the bottlenecks of each hash table, we choose to do experiments with DynaMap and Static Infomap in a dynamic setting. The main reason for this is that DynaMap re-utilizes various structures from previous batches, which should lower the importance of constructing the hash tables. By running Infomap from scratch we can check if a specific scheme provides more benefits for workloads that require frequent hash table construction.

All experiments were performed on a Dell EMC PowerEdge R740 server with two sockets, each having a Xeon Gold 5118 processor with 12 2.30 GHz physical cores with 12 MB of L2 cache per core and 16.5 MB of L3 cache per processor, and 192 GB of main memory. In total, there are 24 physical cores that are able to run 48 threads due to Hyperthreading. The operating system is Ubuntu 24.04 and we compile all codes with gcc-11.4.0 using the -O3 flag. We executed each experiment 10 times. All datasets used were collected from the Stanford Network Analysis Project [Leskovec and Krevl 2014] and are presented in Table 1.

Table 1. Number of vertices and edges of the graphs used in the experiments. All values presented correspond to the ones achieved after transforming directed graphs into undirected ones.

Graph	Vertices	Edges
com-amazon	334,863	925,872
com-dblp	317,080	1,049,866
com-youtube	1,134,890	2,987,624
web-berkstan	685,230	6,649,470

To simulate a dynamic graph, we load a portion of the graph and stream the remainder in batches of fixed size, a common technique used in this area. When experimenting with static Infomap, we execute it from scratch on each batch. When experimenting with DynaMap, we provide an initial set of communities that was generated by a sequential Infomap execution on the initial snapshot and then execute DynaMap on the streamed batches. We could run a parallel static Infomap in the base snapshot, but this could negatively influence results since different parallel executions of Infomap might produce slightly different structures. By giving the same initial structure, we avoid creating unfair circumstances.

We experiment with two initialization scenarios and multiple batch sizes:

- Load 20% of the graph and stream the rest in batches containing 5% of the edges of the original graph.
- Load 90% of the graph and stream 5 batches containing 1%, 0.1%, 0.01%, 0.001% of the edges of the original graph.

To create a stream of edges, we sample 80%/10% of the graph to serve as updates. Since real-world dynamic graphs tend to present fewer deletions than additions [Myers and Leskovec 2014], we create edge streams consisting of 90% additions and 10% deletions. When creating deletions, we select a portion of the sampled edges

and leave them in the initial snapshot to avoid possible errors in deleting an edge that did not exist.

3.1. Performance of Edge Structures

We start our exploration with the hash table that is used to store the edges. To do so, we fix the tables used to store vertices and communities (`cc_hash_table` in this case) and execute the workloads varying the edge hash table. Figure 1 shows the results for the scenario where batches contain 5% of the original graph, while Figure 2 shows the scenarios with batches ranging between 1% and 0.001%. `gp_hash_table` shows a better performance in most scenarios, with `cc_hash_table` usually being no more than 40% slower and sometimes faster for specific batches. The choice of probing function yields similar results, with linear probing being superior in most cases.

To investigate the bottlenecks caused by specific schemes, we measure the time spent on the main sections of DynaMap and Infomap. Figure 3 shows the time spent on pre-processing (DynaMap only), moving vertices across communities, and creating super-graphs. Here we observe that both hash tables provide similar performance when moving vertices, a task that is dominated by find operations. However, during the creation of new super-graphs, `gp_hash_table` shows superior performance.

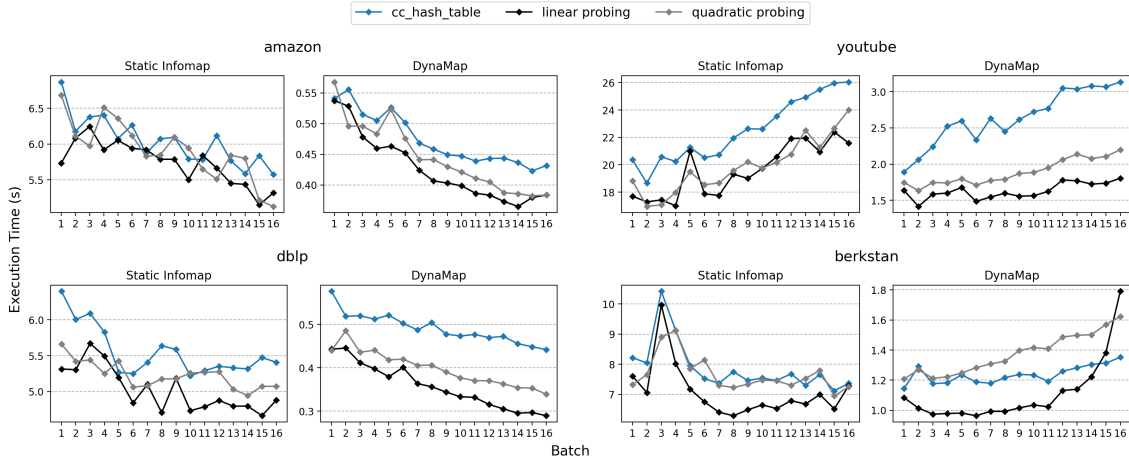


Figure 1. Execution time with 24 threads, per batch, when using different hash tables for the edge structures. We start with 20% of the graph and each batch contains 5% of the original graph.

3.2. Performance of Vertex and Community Structures

Since `gp_hash_table` with linear probing shows superior performance for the edge structure, we use it for the edge hash table when experimenting with the vertex and community hash tables. Given that these experiments have more variations, we opt to simplify the legends in the figures. Table 2 contains the translation for each simplification. Figure 4 shows the results for the scenario where batches contain 5% of the original graph, while Figure 5 shows the scenarios with batches ranging between 1% and 0.001%. Unlike the edge structure, the vertices and communities perform better with `cc_hash_table`. Quadratic probing can be competitive when running DynaMap, but is always worse when running Static Infomap. Linear probing is an interesting case. When representing both

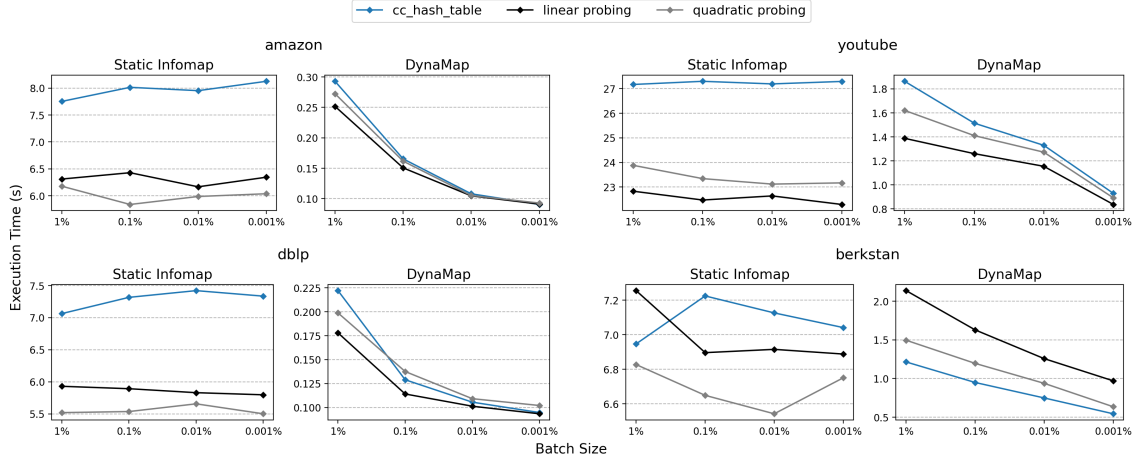


Figure 2. Execution time with 24 threads, per batch, when using different hash tables for the edge structures. We start with 90% of the graph and vary batch sizes between 1% and 0.001% of the original graph.

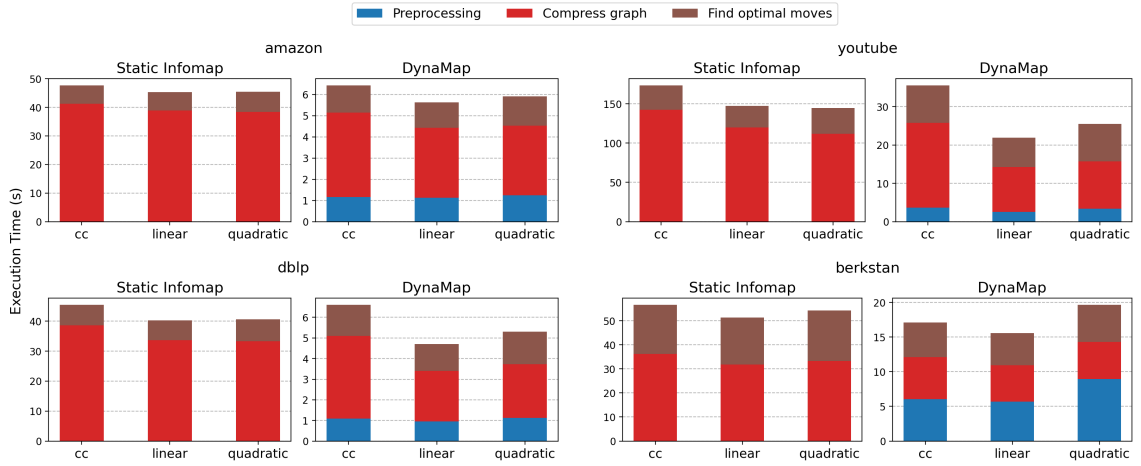


Figure 3. Breakdown of the time spent on the main tasks when using different hash tables for the edge structures. In this case, we start with 20% of the graph and each batch contains 5% of the original graph.

structures, it always performs worse, especially for the 5% batch size scenario of DynaMap (more than 10 times slower for some batches in the larger graphs). However, when representing only vertices, it can be competitive in DynaMap, and when representing only communities, it can be competitive in Infomap.

Figure 6 shows the breakdown of the main tasks of both implementations. We see a similar trend to the edge structure experiments, where performance gains are mainly on the creation of new super-graphs. Again, we highlight the problems of using linear probing for both vertex and community structures. However, unlike other cases where compression is the main bottleneck, linear probing is significantly worse in the find operations performed when moving vertices during DynaMap (over 10 times worse) and in its pre-processing step.

Table 2. Definition of the labels used for the vertex and community figures.

Label	Meaning
cc_all	cc_hash_table for vertices and communities
gp_lin_all	gp_hash_table with linear probing for vertices and communities
gp_quad_all	gp_hash_table with quadratic probing for vertices and communities
gp_lin_v	gp_hash_table with linear probing for vertices and cc_hash_table for communities
gp_quad_v	gp_hash_table with quadratic probing for vertices and cc_hash_table for communities
gp_lin_c	gp_hash_table with linear probing for communities and cc_hash_table for vertices
gp_quad_c	gp_hash_table with quadratic probing for communities and cc_hash_table for vertices

3.3. Memory Consumption

Performance is not the only metric that people take into consideration when choosing a container for their data. The amount of memory required to store such data is also extremely relevant. If a container trades space complexity for performance, it might reduce its viability due to hardware limitations. Therefore, we also analyze the memory consumption of each hash table by verifying the maximum amount of memory used by Infomap and DynaMap during runs with specific configurations.

Table 3 shows the average maximum memory used when starting with 90% of the graph and stream batches of size 1%. We choose to show this specific case for two reasons: we have limited space; and such batch size is large and realistic for real world applications. DynaMap consistently requires less memory than Static Infomap, as it reuses structures across batches and restricts its work to the affected communities. Across all graphs and both algorithms, using cc_hash_table for every structure yields the smallest memory footprint, and every substitution by gp_hash_table increases it. However, the magnitude of this increase depends heavily on which structure is replaced. Switching only the edge structure costs roughly 14–27% more memory, a moderate price given that gp_hash_table is the faster option for this role. When also replacing the community structure, the results yield increases of 30–48% for Static Infomap and 24–28% for DynaMap. Using gp_hash_table for the vertex structure, in contrast, roughly doubles the footprint of Static Infomap and more than triples that of DynaMap on com-youtube (8.00 and 6.51 GB, respectively). Since the vertex-only configuration is nearly indistinguishable from the one that also replaces the community structure, the vertex table is clearly the dominant source of overhead. The choice of probing function has little influence on memory. Quadratic probing is slightly more economical in most configurations, but the difference to linear probing never exceeds about 6%.

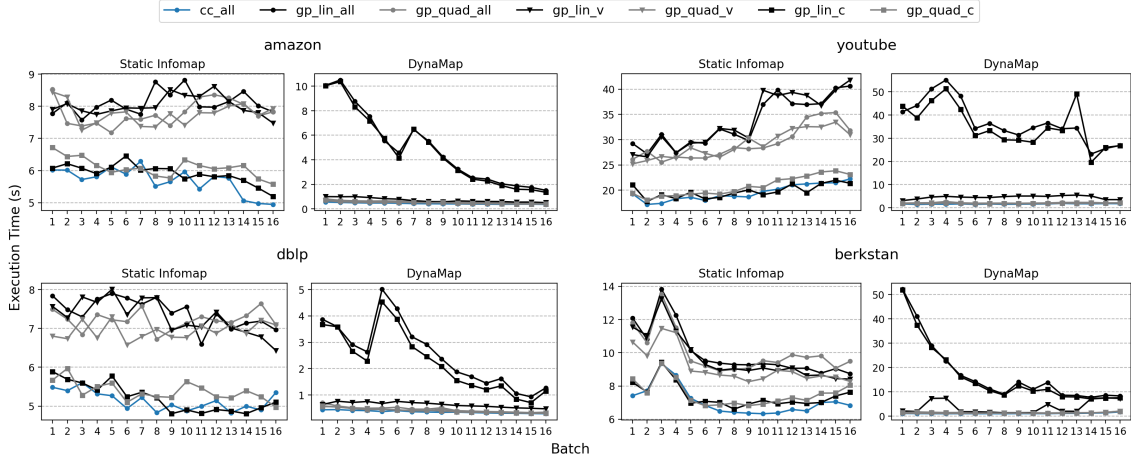


Figure 4. Execution time with 24 threads, per batch, when using different hash tables for the vertex and community structures. We start with 20% of the graph and each batch contains 5% of the original graph.

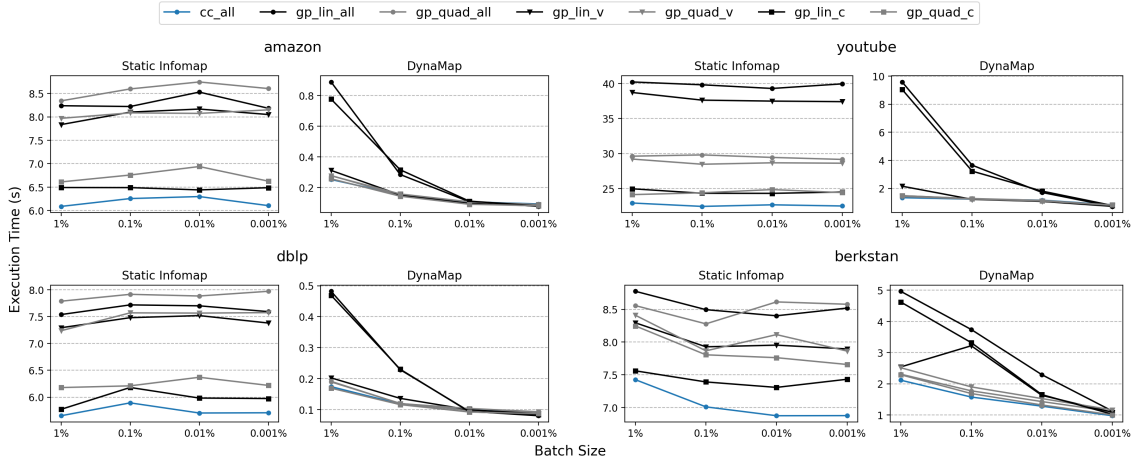


Figure 5. Execution time with 24 threads, per batch, when using different hash tables for the vertex and community structures. We start with 90% of the graph and vary batch sizes between 1% and 0.001% of the original graph.

4. Discussion

The results in Section 3 show a pattern that is easy to state but less easy to anticipate from the container’s design alone: `gp_hash_table` is the better choice for the edge structure, yet `cc_hash_table` is the better choice for the vertex and community structures. This asymmetry suggests that “open addressing beats chaining” or vice versa is not a property of the hash table alone, but of the *access pattern* to which each structure is subjected. The two roles our algorithm places on its hash tables turn out to stress collision resolution in opposite ways, and the breakdown experiments in Figures 3 and 6 make the mechanism behind each result visible rather than leaving it as a black box.

For the edge structure, the gap between hash tables is concentrated in the creation of new super-graphs, not in the find-dominated operations of moving vertices between communities, where all configurations perform comparably. This is consistent with the observation that constructing a super-graph is close to a one-shot bulk build:

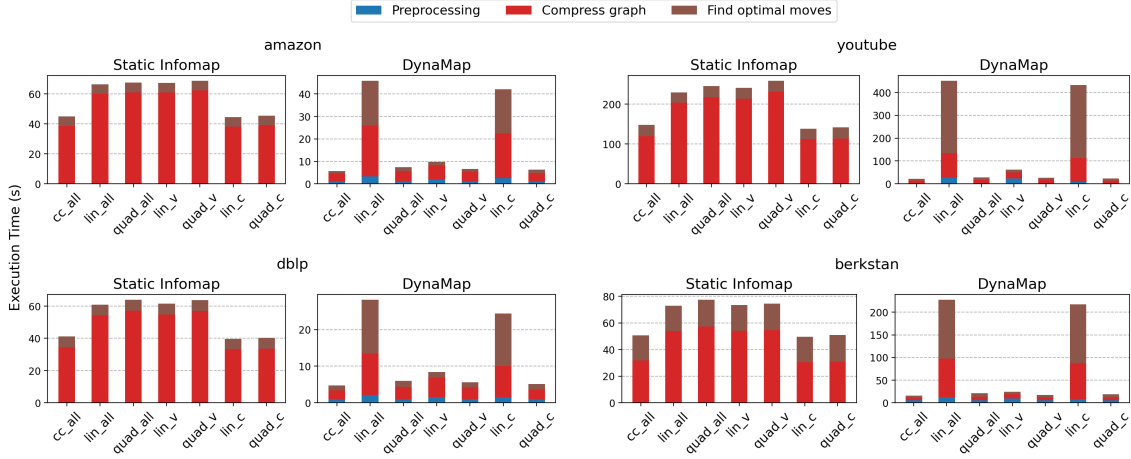


Figure 6. Breakdown of the time spent on the main tasks when using different hash tables for the vertex and community structures. In this case, we start with 20% of the graph and each batch contains 5% of the original graph.

a large number of largely distinct edge keys are inserted in sequence, and duplicates are aggregated rather than repeatedly removed and reinserted. Under this access pattern, `gp_hash_table`'s inline, flat-array storage plays directly to its strengths, avoiding a heap allocation per edge and keeping memory accesses local during a phase that touches a very large number of elements, while the probing strategy itself matters comparatively little because the insertion order does not repeatedly revisit the same regions of the table.

Although the vertex and community structures also impact the super-graph construction, they are more relevant to the movement phase, especially the community structure. As vertices move around, they will start to be concentrated in a subset of communities, which will be accessed and modified more frequently. This skewed pattern exacerbates the primary clustering problem of linear probing policies. DynaMap is more sensitive to this since it already works on a subset of the graph, making linear probing by far the worse option to represent its communities.

Taken together, these results argue for choosing a hash table configuration based on the dominant operation profile of the specific structure being represented, rather than on a single default applied uniformly across an algorithm. This is a finer-grained version of a point made by container-benchmarking work such as BYO [Wheatman et al. 2024], which shows that no single graph container dominates across algorithm classes. Our results show that the same holds *within* a single algorithm, at the level of individual associative structures, whenever that algorithm interleaves a bulk-construction phase (building a super-graph) with an irregular high-access phase (moving vertices between communities). For practitioners designing or tuning similar dynamic graph algorithms, this suggests profiling each auxiliary structure independently before committing to a hash table policy, rather than assuming that a configuration that performs well for one role will transfer to another.

These conclusions should be read against the scope of our evaluation. Our workloads utilize four static SNAP graphs [Leskovec and Krevl 2014] transformed into streams through sampling, rather than graphs that are natively dynamic. Furthermore,

Table 3. Average maximum memory usage (GB) of each configuration for each graph in the 90% scenario with batch size equal to 1%. SI: Static Infomap; DMap: DynaMap; V: Vertices; C: Communities; E: Edges; cc: `cc.hash.table`; lin: Linear Probing; quad: Quadratic Probing.

Config		com-amazon		com-dblp		com-youtube		web-berkstan	
Struct	Hash	SI	DMap	SI	DMap	SI	DMap	SI	DMap
V, C, E	cc	0.82	0.71	0.86	0.64	3.88	1.95	3.31	2.15
E	lin	0.98	0.87	1.02	0.79	4.68	2.47	4.05	2.65
	quad	0.98	0.87	0.99	0.78	4.41	2.33	3.94	2.67
V, C	lin	1.71	1.57	1.74	1.56	8.12	6.51	5.52	4.22
	quad	1.62	1.50	1.84	1.48	7.63	6.14	5.38	4.10
V	lin	1.65	1.56	1.74	1.56	8.00	6.51	5.66	4.21
	quad	1.59	1.52	1.75	1.48	7.61	6.14	5.55	4.09
C	lin	1.24	0.90	1.22	0.80	5.73	2.50	4.32	2.67
	quad	1.22	0.90	1.20	0.81	5.69	2.51	4.31	2.66

all experiments target a single agglomerative community detection algorithm (Infomap) and its dynamic variation (DynaMap) on one multi-socket machine. We also relied on `pb_ds`’s default, mask-based hash-combining policy throughout most experiments; this policy is sensitive to how well the underlying hash function mixes bits when keys are structured or near-sequential, as vertex and super-vertex identifiers tend to be, and we did not isolate this factor from the choice of collision resolution strategy. Future work should vary the hash function and the resize/load-factor policy independently of the probing scheme, and test whether the same observations can be done to other agglomerative or label-propagation-style dynamic community detection algorithms and to genuinely dynamic graphs.

5. Related Work

Although we were unable to find papers that analyze hash table policies for a specific dynamic community detection algorithm, there has been work on evaluating hash tables for static community detection [Kilic et al. 2020] and for general dynamic graph workloads [Wheatman et al. 2024].

`miniVite-x` [Kilic et al. 2020] analyzes the impact that different hash tables have on an implementation of the Louvain algorithm for static community detection [Blondel et al. 2008]. Aside from the clear difference that a static algorithm is being evaluated instead of a dynamic one, the authors only have one relevant hash table to evaluate. Vertices and edges are stored in a CSR format since such structures will not change. The hash table being evaluated is used to store the weight contribution each vertex has towards specific communities, which is created and used during the evaluation of whether a vertex should move. Furthermore, the authors do not test `pb_ds` hash tables.

`BYO` [Wheatman et al. 2024] takes a broader but shallower approach to the same underlying question. It benchmarks over twenty graph containers – including

hash-based general-purpose containers such as `std::unordered_set` and `abseil's flat_hash_set`, and special-purpose containers such as CSR and packed-array – across ten static algorithms and a dynamic batch-insertion workload on many graphs. Unlike `miniVite-x`, `BYO` evaluates hash tables as complete, general-purpose graph containers rather than as auxiliary structures serving a specific role inside an algorithm, and it treats each container as a single fixed configuration rather than varying the collision-resolution strategy or probing policy within one hash table family. Its central finding is that no single container dominates across all algorithm classes. However, the authors do not investigate whether using different containers for different roles can benefit specific algorithms.

Taken together, these two works confirm that hash table choice measurably affects graph workload performance, but neither isolates the effect of `pb_ds's` collision-resolution and probing policies specifically, nor considers a dynamic algorithm that must repeatedly reconstruct auxiliary structures such as super-graphs alongside handling a stream of updates. Our work addresses this gap by evaluating `cc_hash_table` and `gp_hash_table`, with both linear and quadratic probing, as containers for three distinct structures inside a single dynamic community detection algorithm, and by attributing the resulting performance differences to specific phases of that algorithm's execution.

6. Conclusion

In this paper, we presented a series of experiments to evaluate the impacts that hash table policies have over the structures of a dynamic community detection algorithm. To do so, we use two hash table designs from the GNU `libstdc++` Policy-Based Data Structures (`pb_ds`) library – `cc_hash_table` and `gp_hash_table` with both linear and quadratic probing – to represent vertices, communities, and edges. By analyzing the performance yielded by each hash table in different batch size scenarios we conclude that no scheme is superior for all cases. Our experiments show that `cc_hash_table` is the better choice for vertices and communities, while `gp_hash_table` is the better choice for edges in our specific use case. Furthermore, we demonstrate that the main bottleneck lies in constructing the hash table structure.

Our results provide a starting point for further experimentation with different hash table schemes and policies. Although `Infomap` and `DynaMap` present preferences towards representing their structures with specific hash tables, this is not necessarily translated to other graph algorithms. Changes in access patterns might favor schemes and policies that performed poorly for our use case. Furthermore, there are schemes and policies that were not evaluated in this paper. Broader and deeper experimentation is required for making stronger conclusions regarding the use of specific configurations for specific structures inside specific algorithms.

Acknowledgments

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001, and in part by the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq). The authors acknowledge the High-Performance Computing Laboratory of the Pontifical Catholic University of Rio Grande do Sul (LAD-IDEIA/PUCRS, Brazil) for providing support and technological resources, which have contributed to the development of this project and to the results reported within this research.

References

- Blondel, V. D., Guillaume, J.-L., Lambiotte, R., and Lefebvre, E. (2008). Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008.
- Dhulipala, L., Blelloch, G. E., and Shun, J. (2019). Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the 40th ACM SIGPLAN conference on programming language design and implementation*, pages 918–934.
- Ediger, D., McColl, R., Riedy, J., and Bader, D. A. (2012). Stinger: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing*, pages 1–5. IEEE.
- Fan, Z., Cai, Y., Liu, Z., Guo, J., Fan, X., Yang, T., and Cui, B. (2025). Cuckoograph: A scalable and space-time efficient data structure for large-scale dynamic graphs. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*, pages 1265–1277. IEEE.
- Feng, G., Meng, X., and Ammar, K. (2015). Dstinger: A distributed graph data structure for massive dynamic graph processing. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 1814–1822. IEEE.
- Firmli, S. and Chiadmi, D. (2023). A scalable data structure for efficient graph analytics and in-place mutations. *Data*, 8(11):166.
- GNU Project (2024). Policy-based data structures: Design. https://gcc.gnu.org/onlinedocs/libstdc++/manual/policy_data_structures_design.html.
- Green, O. and Bader, D. A. (2016). custinger: Supporting dynamic graph algorithms for gpus. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE.
- Iwabuchi, K., Sallinen, S., Pearce, R., Van Essen, B., Gokhale, M., and Matsuoka, S. (2016). Towards a distributed large-scale dynamic graph data store. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 892–901. IEEE.
- Jaiyeoba, W. and Skadron, K. (2019). Graphtinker: A high performance data structure for dynamic graph processing. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1030–1041. IEEE.
- Kilic, O. O., Tallent, N. R., and Friese, R. D. (2020). Rapid memory footprint access diagnostics. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 273–284. IEEE.
- Leskovec, J. and Krevl, A. (2014). SNAP Datasets: Stanford large network dataset collection. <http://snap.stanford.edu/data>.
- Myers, S. A. and Leskovec, J. (2014). The bursty dynamics of the twitter information network. In *Proceedings of the 23rd international conference on World wide web*, pages 913–924.

- Pandey, P., Wheatman, B., Xu, H., and Buluç, A. (2021). Terrace: A hierarchical graph container for skewed dynamic graphs. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD)*, pages 1372–1385. ACM.
- Rosvall, M., Axelsson, D., and Bergstrom, C. T. (2009). The map equation. *The European Physical Journal Special Topics*, 178(1):13–23.
- Santos, G. G., Lakhotia, K., and De Rose, C. A. (2025). Dynamap: A map equation-based parallel algorithm for detecting communities on dynamic graphs. In *2025 IEEE/SBC 37th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 125–135. IEEE.
- van der Grinten, A., Predari, M., and Willich, F. (2022). A fast data structure for dynamic graphs based on hash-indexed adjacency blocks. In *Proceedings of the 20th International Symposium on Experimental Algorithms (SEA 2022)*, volume 233 of *LIPIcs*.
- Wheatman, B., Dong, X., Shen, Z., Dhulipala, L., Łacki, J., Pandey, P., and Xu, H. (2024). BYO: A unified framework for benchmarking large-scale graph containers. *Proceedings of the VLDB Endowment*, 17:2307–2320.
- Zhang, F., Zou, L., Zeng, L., and Gou, X. (2020). Dolha-an efficient and exact data structure for streaming graphs. *World Wide Web*, 23(2):873–903.